Object Oriented Software Design

Containers

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

October 22, 2010

Outline

Outline

Collections

- A collection (sometimes called a container) is an object that groups multiple objects on a single unit
  - The array is a very simple example of collection
  - In the previous lecture, we have seen the **ArrayList** collection
  - Collections are an essential component of every programming language
  - We will always deal with groups of objects organised in some meaningful way
    - A telephone directory
    - A playing cards' hand
    - A library of books, or a bookshelf
    - all objects in a magazine
    - etc.

Example of container

- Suppose that you define you own class **MyClass**, and you want to put several objects of this class in a dynamically variable array
- Unfortunately, Java arrays have fixed length
  - you define the length of the array at the time of creation, and then you cannot modify it anymore

```
class MyClass {
  void operation() {...}
}
MyClass [] array = new Myclass[10];
...
for (int i=0; i<array.length; i++)
    array[i].operation();
// cannot insert new elements in array
```

- The only way is to create another array with the required size, copy all existing elements of the old array in the new one, and return the new array

Using arrays

```
MyClass[] insertAtEnd(MyClass array[], MyClass elem) {
  MyClass [] temp = new MyClass[array.length+1];
  for (int i=0; i<array.length; i++)
    temp[i]=array[i];
  temp[array.length] = elem;
  return temp;
}
```

- The code above works well, but it can be improved a lot
- Why not encapsulating the above code in a class?

A FIFO Queue

- Suppose we want to implement a simple FIFO queue
- The queue must implement the following interface:
  - **push** inserts an element in the back of the queue
  - **pop** will extract one element from the front of the queue
  - Our queue will contains elements of **MyClass**

MyClass.java

```
public class MyClass {
    private int i;
    static private int counter = 0;

    public MyClass() {
        i = counter++;
    }
    public MyClass(int i) {
        this.i = i;
    }
    public int get() { return i; }

    public String toString() { return "" + i; }
}
```

The FIFOQueue class

- Let's start with the private part

FIFOQueue1.java

```
public class FIFOQueue1 {
    private int capacity = 2;
    private MyClass array[] = new MyClass[capacity];
    private int num = 0;

    private void makeSpace() {
        capacity *= 2;
        MyClass temp[] = new MyClass[capacity];
        for (int i=0; i<num; i++)
            temp[i] = array[i];

        array = temp;
    }
```

- `capacity` is the size of the underlying array
- `num` is the number of elements in the queue
- `makeSpace()` is useful when we need to enlarge the array to insert new elements

The FIFOQueue class

- Now the public interface

FIFOQueue1.java

```
    FIFOQueue1() {}
    public void push(MyClass elem) {
        if (num==capacity) makeSpace();
        array[num++] = elem;
    }
    public MyClass pop() throws EmptyQueueException {
        if (num==0) throw new EmptyQueueException();
        MyClass ret = array[0];
        for (int i=1; i<num; i++) array[i-1] = array[i];
        num--;
        return ret;
    }
    public int getNum() { return num; }

    public int getCapacity() { return capacity; }

    public MyClass getElem(int i) throws IndexOutOfBoundsException {
        if (i<0 || i>=num) throw new IndexOutOfBoundsException();
        return array[i];
    }
```

Using FIFOQueue

FIFOQueue1.java

```
    public static void main(String args[]) {
        FIFOQueue1 myq = new FIFOQueue1();

        for (int i=0; i<5; i++)
            myq.push(new MyClass());

        try {
            System.out.println("Index at 7" + myq.getElem(7));
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Exception: getElem() index out of bound");
        }

        System.out.println("Capacity: " + myq.getCapacity());
        System.out.println("Num: " + myq.getNum());

        try {
            while (myq.getNum() > 0)
                System.out.println("Elem: " + myq.pop().get());
            myq.pop();
        } catch (EmptyQueueException e) {
            System.out.println("Exception: pop() on an empty queue");
        }
    }
```

Beyond MyClass

- Observation: none of the methods of **FIFOQueue** uses anything of the **MyClass** interface
  - Actually, the class **FIFOQueue** could be reused on objects of any type,
  - the code would be very similar, except for the type declaration in the methods
- But how to reuse it?
  - We cannot pass an object different from **MyClass** to the push
- In the old versions of Java (until 1.4), the solution was to make the class contain Objects

The FIFOQueue class

- Let's start with the private part

FIFOQueue2.java

```
public class FIFOQueue2 {
    private int capacity = 2;
    private Object array[] = new Object[capacity];
    private int num = 0;

    private void makeSpace() {
        capacity *= 2;
        Object temp[] = new Object[capacity];
        for (int i=0; i<num; i++)
            temp[i] = array[i];

        array = temp;
    }
```

- Now we have an array of Objects

The FIFOQueue class

- Now the public interface

FIFOQueue2.java

```
    FIFOQueue2() {}
    public void push(Object elem) {
        if (num==capacity) makeSpace();
        array[num++] = elem;
    }
    public Object pop() throws EmptyQueueException {
        if (num==0) throw new EmptyQueueException();
        Object ret = array[0];
        for (int i=1; i<num; i++) array[i-1] = array[i];
        num--;
        return ret;
    }
    public int getNum() { return num; }

    public int getCapacity() { return capacity; }

    public Object getElem(int i) throws IndexOutOfBoundsException {
        if (i<0 || i>=num) throw new IndexOutOfBoundsException();
        return array[i];
    }
```

Using FIFOQueue

FIFOQueue2.java

```
    public static void main(String args[]) {
        FIFOQueue2 myq = new FIFOQueue2();

        for (int i=0; i<5; i++)
            myq.push(new MyClass());

        try {
            System.out.println("Index at 7" + myq.getElem(7));
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Exception: getElem() index out of bound");
        }

        System.out.println("Capacity: " + myq.getCapacity());
        System.out.println("Num: " + myq.getNum());

        try {
            while (myq.getNum() > 0)
                System.out.println("Elem: " + ((MyClass)myq.pop()).get());
            myq.pop();
        } catch (EmptyQueueException e) {
            System.out.println("Exception: pop() on an empty queue");
        }
    }
```

Another problem

- Now the code is more general,
- However when we extract objects (with the `pop()`), we obtain a reference to an **Object**
  - We have to cast the reference to a **MyClass** reference, otherwise we cannot call the `get()` method
  - This is annoying, and it works only if the programmer knows what's inside the FIFOQueue.
- Another problem is that there is no check in the `push()`, we can insert all kinds of objects
  - In particular, we can insert objects of different types (for example Strings, other arrays, etc.)
  - This may be the cause of nasty bugs! See `./examples/09.java-examples/FIFOQueue2Demo.java`
  - In many cases, we want to ensure (at compile time, possibly) that all inserted objects are of the right type

Solution in old Java

- The preferred solution to such problems in old Java was to *wrap* the class inside a different class with a specific interface

```
class FIFOSpecial {
  private FIFOQueue2 myfifo;
  public void push(MyClass elem) {
    myfifo.push(elem);
  }
  public MyClass pop() throws EmptyQueueException {
    return (MyClass)myfifo.pop();
  }
}
```

- This is safe, but it requires the programmer to write annoying extra code
- Generics in Java allow to express these situation without the need to write extra code

Outline

Generic syntax

- Let's see how to implement the **FIFOQueque** class by using generics

FIFOQueue.java

```
public class FIFOQueue<T> {
    private int capacity = 2;
    private Object array[] = new Object[capacity];
    private int num = 0;

    private void makeSpace() {
        capacity *= 2;
        Object temp[] = new Object[capacity];
        for (int i=0; i<num; i++)
            temp[i] = array[i];

        array = temp;
    }
```

Syntax

- The **T** inside angular parenthesis is a type parameter
  - It says that this class is parametrised by the type **T**
  - You can also use the class without parameters (this is called *raw type*)
  - However, the most common and clean use is to assign **T** a type when the class is used
- Relationship between types
  - **FIFOQueue**<**Integer**> is a different type than **FIFOQueue**<**MyClass**>
  - There is no relationship between the two types
- Notice that the array is still an array of **Object**
  - unlike C++, it is not possible to declare an array of **T**, we will see later why

Public interface

- Now the public interface

FIFOQueue.java

```
    FIFOQueue() {}
    public void push(T elem) {
        if (num==capacity) makeSpace();
        array[num++] = elem;
    }
    public T pop() throws EmptyQueueException {
        if (num==0) throw new EmptyQueueException();
        T ret = (T)array[0];
        for (int i=1; i<num; i++) array[i-1] = array[i];
        num--;
        return ret;
    }
    public int getNum() { return num; }

    public int getCapacity() { return capacity; }

    public T getElem(int i) throws IndexOutOfBoundsException {
        if (i<0 || i>=num) throw new IndexOutOfBoundsException();
        return (T)array[i];
    }
```

Using FIFOQueue

FIFOQueue.java

```
    public static void main(String args[]) {
        FIFOQueue<MyClass> myq = new FIFOQueue<MyClass>();
        FIFOQueue<String> stq = new FIFOQueue<String>();

        for (int i=0; i<5; i++) myq.push(new MyClass());
        for (int i=0; i<5; i++) stq.push("string number " + i);

        try {
            while (myq.getNum() > 0 && stq.getNum() > 0)
                System.out.println("Elem: " + myq.pop().get() + " string:
            myq.pop();
        } catch (EmptyQueueException e) {
            System.out.println("Exception: pop() on an empty queue");
        }
    }
```

A more efficient FIFOQueue

- The previous implementation of the **FIFOQueue** is inefficient
  - when we `pop()` an element, we have to go through the whole array and move all references one step back
  - If the array contains **n** elements, this requires **n** assignment operations
  - Also, when we `push()` inside an array that is already full, we have to copy all references in a new array, and again this requires **n** operations
- Let's now see a different implementation based on the concept of dynamic list

An efficient queue

- We define a class **Node** to contain each element

EfficientQueue.java

```
public class EfficientQueue<T> {
    class Node {
        T elem;
        Node next = null;
        Node prev = null;
        Node(T elem) { this.elem = elem; }
    }
    private Node head = null, tail = null;
```

- We implemented Node as an inner class
  - It is possible to define classes inside other classes
  - This class is not part of the interface, but only of the implementation: therefore it is not declared as public (it is *package public*)
  - It can be used inside the class without further specification
  - It can be referred by other classes inside the package as **EfficientQueue.Node**

Inner classes

- Of course, it is also possible to define public classes inside other classes, they will be part of the class interface
- The inner class can also be declared private, if you do not want to put it in the interface of the class
- You can use the type parameter **T** inside **Node**
- A method of node can use all the members of class EfficientQueue inside its methods (we do not use this feature in this specific case)
  - Basically, every object of the inner class has an hidden reference to the object of the outer class that created it
  - For example, a method of **Node** could access member `head`

Insertion of a new element

- To insert a new element, we first create the corresponding Node, and then we *link* the node to the correct place (the tail of the queue)

EfficientQueue.java

```
    public void push(T elem) {
        Node n = new Node(elem);
        n.prev = tail;

        if (tail != null) tail.next = n;
        else head = n;
        tail = n;
    }
```

Extraction of an element

- To extract an element, we *unlink* the node from the head, and adjust all references

EfficientQueue.java

```
    public T pop() throws EmptyQueueException {
        if (head == null) throw new EmptyQueueException();
        Node n = head;
        head = n.next;
        if (head != null) head.prev = null;
        else tail = null;
        return n.elem;
    }
```

- The usage is pretty similar to the one of **FIFOQueue**
- A simple exercise: add an internal counter `num` of elements, and a method `getSize()` that returns the number of elements

Inheritance

- What if we want to insert elements of different type that have a common base class?
  - For example, we may want to insert **Instrument**s inside our **FIFOQueue**
  - Everything works as expected

InstrumentQueue.java

```
        EfficientQueue<Instrument> myq =
            new EfficientQueue<Instrument>();
        myq.push(new Woodwind());
        myq.push(new Violin());
        try {
            while(true) myq.pop().play(Note.C);
        } catch (EmptyQueueException e) {
            System.out.println("I played all instruments");
        }
```

Visiting the container

- Before continuing, let's add another functionality to our queue. It is now time to call it **QueueList**
- We want to be able to visit all elements of the list one by one
- For example, we would like to print all elements inside the queue
- We need some way to get the first element, and then move to the following ones, in sequence
- The first approach is:
  - a have a method that initialise the visit,
  - and a method to get the next unread element
  - a method to check if there are unread elements

QueueList code

QueueList.java

```
    public void start() { curr = head; }

    public boolean hasNext() {
        if (curr == null) return false;
        else return true;
    }

    public T getNext() throws NoNextElementException {
        if (curr == null) throw new NoNextElementException();
        T ret = curr.elem;
        curr = curr.next;
        return ret;
    }
```

How to use QueueList

- This is how to use the new interface:

QueueList.java

```
        QueueList<MyClass> myq = new QueueList<MyClass>();

        myq.push(new MyClass());
        myq.push(new MyClass());
        myq.push(new MyClass());
        myq.push(new MyClass());

        myq.start();
        try {
            while (myq.hasNext())
                System.out.println(myq.getNext().get());
        } catch(NoNextElementException e) {
            System.out.println("empty!");
        }
```

Limitations

- The new interface works well when we have to explore the queue with one single index
  - However, suppose we need two indexes at the same time (for example, when we have two nested loops on the same list)
  - And what if we need three indexes?
  - Also, we may want to go back and forth in the sequence
- A more general solution is to separate the index from the class
  - In this way, the sequence is separated from the way we visit it
  - The *index* in this case is called **iterator**

Outline

Iterator interface

- What is the interface of an iterator?
  - We must be able to retrieve the element corresponding to the iterator, and move the iterator forward
  - We must be able to understand when the iterator has reached the end of the sequence
  - Optionally, we must be able to remove an element from the list
- The following interfaces are in the *collection framework* of Java:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}

public interface Iterable<E> {
  Iterator<E> iterator();
}
```

Implementation in QueueListIt

- We first show to class private section:

QueueListIt.java

```
import java.util.*;

public class QueueListIt<E> implements Iterable<E> {
    private class Node {
        E elem;
        Node next = null;
        Node prev = null;
        Node(E elem) { this.elem = elem; }
    }
    private Node head = null, tail = null;
```

- Notice that our class now implements **Iterable**, so we have to provide method `iterator()` that creates a new iterator that refers to the beginning of the sequence

Iterator Implementation

- The iterator is a private inner class

QueueListIt.java

```
    private class QLIterator implements Iterator<E> {
        private Node curr = null;
        private Node prev = null;

        QLIterator() { curr = head; }

        public boolean hasNext() {
            if (curr == null) return false;
            else return true;
        }

        public E next() {
            if (curr == null) return null;
            E elem = curr.elem;
            prev = curr;
            curr = curr.next;
            return elem;
        }
```

Removing an element

- This is the remove method, enforced by the **Iterator** interface

QueueListIt.java

```
        public void remove() {
            if (prev == null) return;
            // remove element
            Node p = prev.prev;
            Node f = prev.next;
            if (p == null) head = f;
            else p.next = f;
            if (f == null) tail = p;
            else f.prev = p;
            prev = null;
        }
```

The QueueListIt class

- The only method that needs to be added is `iterator()`

```
public Iterator<E> iterator() { return new QLIterator(); }
```

The for-each statement

- From Java 5 we have one new statement that can be used only with collections (i.e. classes that implement the **Iterable** interface
  - Here is an example in **QueueListIt**

QueueListIt.java

```
        QueueListIt<MyClass> myq = new QueueListIt<MyClass>();

        for (int i=0; i<6; i++)
            myq.push(new MyClass());

        System.out.println("Elements:");
        for (MyClass c : myq)
            System.out.println(c);

        Iterator<MyClass> i = myq.iterator();
        while (i.hasNext())
            if (i.next().get() % 2 == 0)
                i.remove();
```